第10章 名字控制

本章概要

- 创建名字是程序设计过程中的一项基本活动，当一个项目很大时，它将不可避免地包含大量的名字。
- C++允许我们对名字的产生和名字的可见性进行控制，包括这些名字的存储位置以及名字的连接

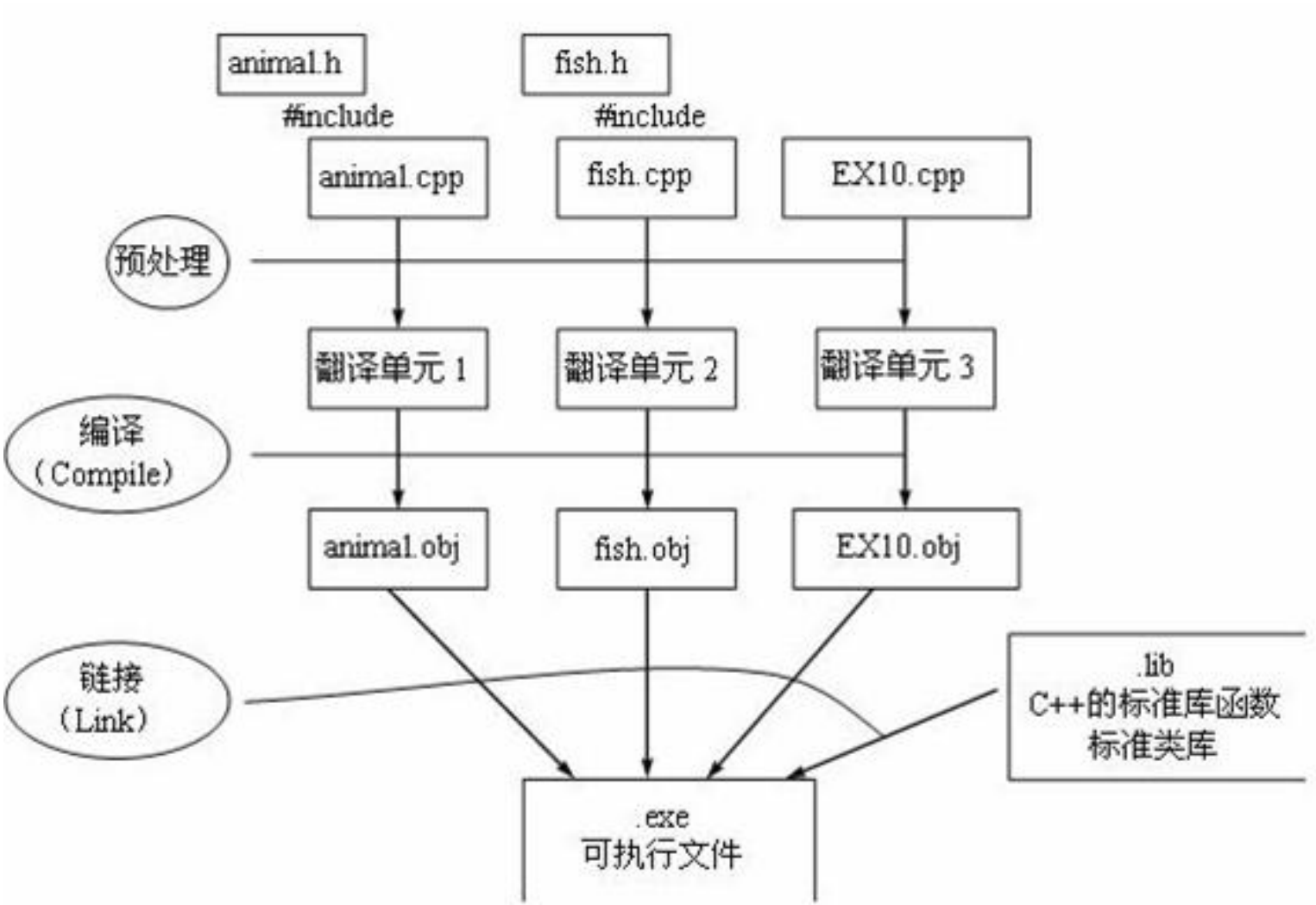
静态元素

- 静态变量
- 静态对象

- 特点：
  - 保存位置：静态存储区
  - 作用域：仅在翻译单元内部
  - 在main函数执行前被创建；
  - 仅第一次调用时初始化；

C10:StaticDestructors.cpp；

C++的连接类型


animal.h
fish.h
#include
#include
animal.cpp
fish.cpp
EX10.cpp
预处理
翻译单元 1
翻译单元 2
翻译单元 3
编译
(Compile)
animal.obj
fish.obj
EX10.obj
链接
(Link)
.lib
C++的标准库函数
标准类库
.exe
可执行文件

连接类型

- 在C++中，对象，函数，类型，模板都有名字，所有这些有名字的元素，都会涉及到连接性。
- 所有这些元素都定义在翻译单元中，在C++中，一个翻译单元通常是一个.cpp文件，准确地说，应该是一个经过预处理后的.cpp文件。
- 连接性就是用来区分该对象是仅仅在该翻译单元内可见，还是对于所有的翻译单元都可见。以此区分是内部连接(internal linkage)还是外部连接(external linkage) 。

连接类型

- 对于采用内部连接的名字，由于是局部于该翻译单元的，所以可以在其他单元中使用同样的名字，而不必担心冲突。
- 而对于需要与其他翻译单元共享的存储空间、类型等元素，需要定义为外部连接。

内部连接

```
static void f(); //a static function
static int q; //a static object declared in global scope
namespace //members of anonymous namespace
{
        class C{};
        int x;
 }
const int M=1000; //const object not declared extern
union{ //members of an anonymous union
        int x;
        float y;
};
```

外部连接

```
int n; //global non-static, hence external linkage
class C
{
        void f(); // member functions
        static int n;// static data members
};
extern const K; //defined in a different translation unit
void func ();
namespace NS
{
        class D{}; // qualified name NS::D has external linkage
}
enum DIR
{
        Up,
        Down
} // DIR, Up, and Down have external linkage
```

替代连接说明

- 如果在C++中编写一个程序需要用到C的库，并且以以下的形式声明：
  - float f(int a, char b);
- 连接器将抱怨无法找到该函数。因为C++的编译器将会把这个函数的名字替换为_f_int_char
- 如果不希望编译器进行上述的转换，则使用
  - extern “C” float f(int a, char b);

Calling C from C++

- Use **extern "C"** to tell the C++ compiler to use C calling conventions

```
// calling C function from C++:

extern "C" double sqrt(double);      // link as a C function

void my_c_plus_plus_fct()
{
        double sr = sqrt(2);
        // …
}
```

名字空间(namespace)

- C++的名字空间，把一个全局名字空间分解成多个可管理的小空间。
  - 创建名字空间
  - 使用名字空间

名字空间

- 我们需要一种明确的方法，将元素限定在某个作用域内有效；
- 空间是一种层次化的命名方法，这也是我们给大多数事物定位所采取的手段。
- 考虑下面的例子：
  - 门牌号码
  - 域名
- 在没有名字空间的情况下，C库的提供者往往采取冗长、不方便的名字给自己的函数命名。随着系统的复杂性不断增加，这种方法受到的限制也越来越大。

创建名字空间

- 下面的代码产生了一个新的名字空间，在定义部分可以包含各种定义，如类，结构，函数等，这些定义都包含在这个名字空间内。

```
namespace MyLib {
  // Declarations
}
int main() {} ///:~
```

创建名字空间

- 创建名字空间与创建一个类非常相似。如同class, struct, enum和union一样，都是把它们的成员的名字放到了不同的空间，而不同之处是，名字空间的唯一目的是产生一个新的定位某个元素的路径。

创建名字空间

- 尽管名字空间和class, struct, enum和union看上去很类似，但它们之间有如下的区别：
  - 名字空间只能在全局范围内定义，它不能定义在class, struct, enum和union中。但名字空间它们自己可以相互嵌套；
  - 名字空间的结尾处不必跟一个分号；
  - 不能像创建类一样创建一个名字空间的实例；

创建名字空间

- 可以按类的语法来定义namespace，定义的内容在多个头文件中延续，就好像重复定义这个namespace一样，效果等同与这两处的定义合并为一个名字空间；

```
//：C10: Header1.h
namespace MyLib {
  extern int x;
  void f();
  // ...
}
```

```
//: C10:Header2.h
namespace MyLib { // NOT a redefinition!
  extern int y;
  void g();
  // ...
}
```

创建名字空间

- 重命名名字空间
  - 可以指定一个别名，这样就不必敲打那些开发商提供的冗长的名字了。

```
namespace BobsSuperDuperLibrary {
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
int main() {} ///:~
```

创建名字空间

- 每个翻译单元都可包含一个未命名的名字空间，这个名字空间中的元素在该翻译单元有效。
- 在一个翻译单元中可以出现多个匿名namespace，并且相同层次的匿名namespace实际上被合成为同一个；出现在不同翻译单元的匿名namespace中的相同标识符相互独立不会发生冲突，因此我们可以把那些只希望在同一个翻译单元范围可见的全局标识符放入一个匿名namespace中，效果与前面加static相同；

使用名字空间

- 有三种引用名字空间内名字的方法：
  - 作用域运算符
  - 使用指令(using directive)
    - 将所有名字引入到名字空间内
  - 使用声明(using declaration)
    - 指定引入特定的名字

使用名字空间

- 使用域运算符
  - 名字空间内的任何名字都可以用作用域运算符做明确的指定，就像引用一个类中的名字一样。

使用名字空间

- 使用域运算符
  - C10:ScopeResolution.cpp

```
namespace X {
        class Y {
              static int i;
        public:
             void f();
        };
        class Z{
              int u, v, w;
         public:
              Z(int i);
              int g();
        }
        void func();
}
```

```
int X::Y::i = 9;

X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }

void X::func() {
  X::Z a(1);
  a.g();
}
```

使用名字空间

- 使用指令(using directive)将所有名字引入到名字空间内，这样，接下来的代码中就无需使用完整的限定。

```
//:NamespaceInt.h
//:NamespaceMath.h
//:Arithmetic.cpp
```

使用名字空间

- 使用using指定引入特定的名字，也称为使用声明(using declaration)
  - Using declaration的优先级比using directive高。

    //C10:UsingDeclaration.h

    //C10:UsingDeclaration1.cpp
- 可以通过使用声明将一个名字空间里的名字导入另一个名字空间

  //C10:UsingDeclaration.h

  //C10:UsingDeclaration2.c

```
#include "UsingDeclaration.h"
void h() {
  using namespace U; // Using directive
  using V::f; // Using declaration
  f(); // Calls V::f();
  U::f(); // Must fully qualify to call
}
int main() {} ///:~
```

使用名字空间

- 注意：不要轻易将一个全局的using指令用在头文件中。如果确实需要在头文件中使用using，尽可能将其局部化，也就是：
  - 尽可能使用using declaration引入特定的名字；
  - 或在一个特定的范围内使用using directive；

名字空间：小结

- C++的名字空间特征，把一个全局名字空间分解成多个可管理的小空间。
  - 创建名字空间
    - 定义名字空间
    - 使用别名
  - 使用名字空间
    - 用作用域运算符
    - 用使用指令(using directive)将所有名字引入到名字空间内
    - 用使用声明(using declaration)指定引入特定的名字

与Java的Package的比较

- 创建名字空间
  - 名字空间与文件系统的关系
- 使用名字空间
- Java中的 import static

静态元素

- 全局静态
  - 全局静态变量
  - 全局静态函数
- 函数中的静态变量
- 类中的静态成员
  - 静态成员数据
  - 静态成员函数
- 静态数据的初始化依赖

静态的含义

- 静态有两种含义
  - 声明为静态的元素是在固定的地址上进行存储分配，也就是说对象是在一个特殊的静态数据区上创建的。
  - 声明为静态的元素对一个特定的编译单位来说是局部的。这样，static控制这个名字的可见性：这个名字在该翻译单元之外是不可见的。

函数中的静态变量

- 例： C10:StaticVariablesInFunctions.cpp
- 这个例子在说明函数内静态变量的同时，也说明了它的危险性；

函数中的静态对象

- 如果函数内部定义的是用户自定义的静态对象，便涉及到构造函数和析构函数的调用。
- 函数内静态对象的构造函数在这个函数第一次被调用时构造并初始化。如果该函数没有被调用，则不会被初始化。
  - 没有被初始化是否意味着不占用内存？
- 函数内静态对象的析构函数在程序从main中退出时调用。析构函数调用的次序按照初始化相反的顺序进行。
- 如果静态对象没有初始化过，则不会调用析构函数。

//C10:StaticDestructors.cpp

类中的静态成员

- 静态数据成员
  - 类中普通的数据成员，对于每个类的实例，都占据各自的存储空间。或者说，类的每个实例之间没有共享数据。
  - 如果类的实例之间需要共享数据，则需要静态数据成员。

类中的静态成员

- 下面的例子中，instanceCount用来记录一个类实例化的次数。

```
//C10:ClassCount.h
class ClassCount
{
        static int instanceCount;
public:
        ClassCount();
        virtual ~ClassCount();
};
//C10::ClassCount.cpp
int ClassCount::instanceCount = 0;
ClassCount::ClassCount()
{
        instanceCount++;    }
```

静态成员的初始化

- 静态成员需要初始化代码，用于分配存储空间，这部分的代码可以看作是类定义的一部分。
- 一旦被定义，用户就不能重新定义它。
- 回顾：static const修饰符的含义 。
- 静态数组的初始化。
  - C10: StaticArray.cpp

静态成员函数

- 静态成员函数是为类的全体对象服务而不是为了一个类的具体的实例服务。
- 类的静态成员类似与一个在该类的命名空间下的全局函数。

```
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

静态成员函数

- 静态成员函数不能访问非静态的数据成员和成员函数，因为他没有this指针。
- 一个静态数据成员和静态成员函数在一起使用的例子：
  - C10:StaticMemberFunctions.cpp
- 使用静态数据成员和静态成员函数完成单实例模式
  - C10:Singleton.cpp

静态对象初始化的依赖问题

- 在一个指定的翻译单元中，静态对象的初始化顺序严格按照对象在该单元中定义出现的顺序。而清除的顺序正好相反。
- 对于分布在多个翻译单元中的静态对象而言，无法保证严格的初始化顺序，可能引起一些问题。

静态/全局对象初始化的依赖问题

- 第一个文件：

```
#include <fstream>
Std::ofstream out(“out.txt”);
```

- 第二个文件：

```
#include <fstream>
extern std::ofstream out;
class Oof{
public:
   Oof(){out << “ouch”;}
};
Oof oof;
```

解决方案

- 使得初始化的顺序由程序来控制，而不是让连接器“随机”决定。
- 具体的方法：利用函数的静态变量在函数第一次调用时初始化的性质。
  - C10:Dependency1.h
  - C10:Dependency2.h
  - Technique2.h
- 对于任何初始化依赖因素来说，可以把一个静态对象放在一个能返回对象引用的函数中。这样，访问静态对象的唯一途径就是调用这个函数。如果该静态对象需要访问其它依赖于它的对象时，就必须调用那些对象的函数。静态初始化的正确顺序由设计的代码保证。

小结：静态元素

- 全局静态
  - 全局静态变量
  - 全局静态函数
- 函数中的静态变量
- 类中的静态成员
  - 静态成员数据
  - 静态成员函数
- 静态数据的初始化依赖